The Nature and Origin of Rational Errors in Arithmetic Thinking: Induction from Examples and Prior Knowledge

TALIA BEN-ZEEV
Yale University

Students systematically and deliberately apply rule-based but erroneous algorithms to solving unfamiliar arithmetic problems. These algorithms result in erroneous solutions termed rational errors. Computationally, students' erroneous algorithms can be represented by perturbations or bugs in otherwise correct arithmetic algorithms (Brown & VanLehn, 1980; Langley & Ohlson, 1984; VanLehn, 1983, 1986, 1990; Young & O'Shea, 1981). Bugs are useful for describing how rational errors occur but bugs are not sufficient for explaining their origin. A possible explanation for this is that rational errors are the result of incorrect induction from examples. This prediction is termed the "induction hypothesis" (VanLehn, 1986). The purpose of the present study was to: (a) expand on past formulations of the induction hypothesis, and (b) use a new methodology to test the induction hypothesis more carefully than has been done previously. The first step involved teaching participants a new number system called NewAbacus, a written modification of the abacus system. The second step consisted of dividing them into different groups, where each individual received an example of only one part of the NewAbacus addition algorithm. During the third and final step, participants were instructed to solve both familiar and unfamiliar types of addition problems in NewAbacus. The induction hypothesis was supported by using both empirical and computational investigations.

The author wishes to thank Robert Sternberg, Michael Tarr, Richard Gerrig, Mark Ashcraft, Kurt VanLehn, and an anonymous reviewer, for their extremely helpful comments on earlier versions of this work. Special thanks are given to Henry Kaufman who provided invaluable help with computationally modeling the NewAbacus number system.

The research reported in this article was supported by a graduate fellowship from Yale University, as well as by a grant under the Javits Act program given to Robert Sternberg (Grant #R206R00001) as administered by the Office of Educational Research and Improvement, U.S. Department of Education. The findings and opinions expressed in this study do not reflect the positions or policies of the Office of Educational Research and Improvement or the U.S. Department of Education.

Correspondence and requests for reprints should be sent to Talia Ben-Zeev, Department of Psychology, Yale University, P.O. Box 208205, New Haven, CT 06520-8205. Email: Talia@yalevm.ycc.yale.edu.
When students commit errors on arithmetic problems, they often do so as a result of actively constructing erroneous rules or algorithms (Ashlock, 1976; Brown & VanLehn, 1980; Cox, 1975; Lankford, 1972; VanLehn, 1983). These erroneous algorithms are rule-based, and are applied systematically and deliberately, rather than randomly. That is, students are correctly following incorrect rules, rather than incorrectly following correct ones. By doing so, students produce erroneous solutions termed rational errors.

The term rational error is an oxymoron. It suggests a contradiction between a student’s rational approach during the problem solving process and the erroneous results the student produces. On the one hand, the student is a creative and active problem-solver, who not only invents rules but follows them as well. On the other hand, the rules the student creates are often flawed. What then goes “wrong” in the problem solving process? Is it possible to localize the exact point that corresponds to the inception of an erroneous algorithm? What reasoning mechanisms are responsible for its derivation? Pursuing these and similar questions allows us to investigate the ways students produce rational errors in particular, and contributes to the understanding of fundamental mental processes underlying arithmetic learning and problem solving in general.

The study of rational errors is also important from an educational standpoint. Knowledge of how students produce rational errors can be used as a basis for guiding them toward developing correct problem-solving skills. This study, however, focuses on theoretical questions regarding the nature and origin of rational errors. It is believed that a theoretical account of rational errors helps build a foundation for applied educational investigations in the area of mathematical thinking.

The current study’s approach to studying the nature and origin of rational errors offers both theoretical and methodological advantages. On the theoretical front, it suggests that “what goes wrong” in the production of rational errors is a result of inductive reasoning processes that operate on incomplete knowledge. That is, when students encounter a novel problem they do not know how to solve, they overgeneralize or overspecialize rules from familiar examples in the arithmetic domain in order to produce a solution. This idea is termed the “induction hypothesis” (VanLehn, 1986). The current study improves on past formulations of the induction hypothesis by suggesting that induction is not only based on current examples a student is exposed to in the process of learning a particular arithmetic skill (e.g., subtraction), but can also be traced to prior knowledge algorithms in different domains (e.g., addition), as well (for a discussion on how prior knowledge plays an important role in facilitating acquisition of procedures see Ohllson & Rees, 1991).

On the methodological front, this study offers a combined experimental and computational investigation. The controlled experimental component is
new to an area dominated by computational approaches. Experimental control is achieved by teaching participants a new number system, and then determining which particular examples of an arithmetic algorithm participants learn. This procedure makes it possible to attempt to trace participants' rational errors on novel problems to the examples they received. In particular it allows one to determine whether receiving a set of specific examples causes participants to generate particular types of errors. Thus, controlling which examples a participant receives allows one to determine better the role of examples as possible sources of induction. This methodology is advantageous, because until now researchers have collected student responses to arithmetic problems, and proceeded to analyze them computationally, without having precise control over which examples a student was exposed to in the learning process (Brown & VanLehn, 1980; Langley & Ohlsson, 1984; VanLehn, 1983, 1986; Young & O’Shea, 1981). Before introducing the specifics of the present study’s approach, however, I will present an overview of research in this area.

THE NATURE OF RATIONAL ERRORS

A rational error results from a perturbation or bug in the otherwise correct algorithm. For example, in the domain of subtraction, students commonly produce what is known as the Smaller-From-Larger bug (VanLehn, 1983). This bug occurs on subtraction problems where the bottom digit (B) is larger then the top digit (T). Instead of borrowing a ten from the next digit to the left of T, the student who uses the Smaller-From-Larger bug simply subtracts T from B by swapping the top and bottom digits. Thus, the student never borrows. In each column he or she proceeds to subtract the smaller digit from the larger one, regardless of the respective positions of the two digits. The following is an illustration of the Smaller-From-Larger bug:

\[
\begin{array}{c}
63 \\
-29 \\
\hline
46
\end{array}
\]

Several computational models have been developed to simulate and better understand the nature of rational errors (Brown & VanLehn, 1980; Langley & Ohlsson, 1984; VanLehn, 1983; 1986; Young & O’Shea, 1981). Their main tenets are presented next.

Young and O’Shea’s Production-System Model of Rational Errors

Young and O’Shea (1981) offer a production-system model for simulating students’ subtraction errors. A production-system is made up of rules of the form, IF C THEN A (C ⇒ A), where C is the condition and A is the action
(for applications of production-systems to other learning domains see Klahr, Langley, & Neches, 1987). Young and O'Shea first used the condition-action rules to model correct subtraction. Then, in order to simulate how students produce rational errors, they added or omitted rules from the correct production-system. For instance, they modeled the Smaller-From-Larger bug by omitting a rule that triggers the borrowing procedure.

Young and O'Shea's production-system approach is a useful tool for describing a fine-grained step-by-step analysis of a student's problem-solving process. However, as Young and O'Shea themselves admit, it may be criticized for being *ad hoc*. That is, the production-system is designed maximally to match the student's responses after the student has already produced them. There is no theoretical motivation for why certain rules are added to or deleted from the correct algorithm in order to simulate the student’s production of rational errors. These rules are chosen because they best match student performance.

**Brown and VanLehn's Repair Theory:**

A Principled Approach to the Study of Rational Errors

In contrast, repair theory is a theory-driven computational model of rational errors. It was first offered by Brown and VanLehn (1980) and later expanded on by VanLehn (1983, 1990). Repair theory suggests that a student initially masters a *prefix* of the subtraction algorithm. A prefix corresponds to the current instructional segment of the subtraction curriculum (e.g., how to borrow from a nonzero digit), as well as segments leading up to it (e.g., how to subtract without borrowing). This prefix makes up the student’s *core procedure* for subtraction. That is, the core procedure is the student's cognitive representation of the prefix.

For example, a student may only perfectly master the steps for borrowing from a nonzero digit. Thus, when the same student attempts to solve a problem that requires borrowing across zero, he or she will reach an *impasse*. That is, the student will reach a point in the problem-solving process where his or her core procedure lacks the rules to continue. To overcome the impasse, the student chooses a *repair* from a set of heuristics. A repair is a set of actions that modifies the core procedure, and gets it “unstuck.” A repair is thus a process that results in a rational error or a bug. For example, the student may choose to apply a repair that causes the Stops-Borrow-At-Zero bug, as shown:

\[
\begin{align*}
305 \\
-109 \\
206
\end{align*}
\]

The student who uses the Stops-Borrow-At-Zero bug correctly adds a ten to the current top digit, but fails to borrow a ten from the top digit to its
left. The idea behind repair theory is that bugs are a result of a cross-product forming all possible combinations between sets of impasses and repairs.

It is argued that the term repair is a misnomer (see also Maurer, 1987), because it implies correctly fixing or rectifying an incomplete procedure. In repair-theory language, however, a repair refers to the process of creating an algorithmic error, or bug. Furthermore, there is no clear term for the incorrect answer the student produces. I thus suggest that the term rational error be used to describe the results of executing a buggy procedure. The new term clarifies the distinction between the actual perturbation in the student's algorithm (i.e., bug), the heuristic that created the perturbation (i.e., repair), and the final erroneous answer the student produces (i.e., rational error). Note that the term "rational" is preferred over "systematic" because it describes the rule-based origin of the error without reference to the error's stability across problems.

Brown and VanLehn's approach is principled because it describes the psychological process by which a rational error is created. It gives a concise description of errors, due to the fact that it can explain a whole set of similar bugs as manifestations of the same impasse crossed with the set of repairs. It can thus predict a priori the performance of students over a large range of problems, rather than describing their rational errors in an ad hoc fashion.

Although specifying perturbations in otherwise correct algorithms provides a satisfying description of the nature of rational errors, it does not provide satisfactory answers concerning their origin. As VanLehn (1986) argues: "It is important to stress that bugs are only a notation for systematic errors and not an explanation." What, then, is the origin of students' rational errors?

THE ORIGIN OF RATIONAL ERRORS: THE INDUCTION HYPOTHESIS

VanLehn (1986) suggests that most rational errors are induced from the instructional prefix. That is, prefixes are either overspecialized or overgeneralized to form students' rational errors. An example of overgeneralization is the bug N-N-Causes-Borrow. This bug is illustrated as follows:

\[
\begin{array}{c}
7 \\
8 \quad 1 \\
- 3 \quad 4 \\
\hline
4 \quad 1 \quad 0
\end{array}
\]

The student who has the N-N-Causes-Borrow bug has mastered the prefix which says to borrow when \( T < B \), and not to borrow when \( T > B \). When
the student encounters a novel problem where $T = B$, he or she overgeneralizes the rule "borrow when $T < B$" to become the rule "borrow when $T \leq B$." A student can also overspecialize a rule from a known prefix. For example, a student who has only been shown how to borrow on two-column problems may reason that a borrowing action only occurs in the units column.

VanLehn takes two computational approaches to showing that rational errors are induced from examples. First, he examines textbooks and lesson plans in order to identify the different instructional prefixes in the subtraction curriculum. For example, VanLehn decides to incorporate the *borrow across zero* procedure in his set of prefixes for subtraction, because textbooks devote a whole instructional segment to teaching students this skill. Finally, VanLehn shows that students' rational errors can be modeled by overgeneralizing and overspecializing rules from individual examples in the instructional prefixes. By using this approach, however, VanLehn can explain only 33% of second- through fifth-grader's rational errors as stemming from induction.

VanLehn attempts to support the induction hypothesis by performing a second, and a more "liberal" analysis, as he himself terms it. To simulate induction of rational errors, he modifies the correct procedure for subtraction by introducing rules that build on visual-numerical features of subtraction problems (e.g., top-of, left-of, bottom-equal-0, etc.). The logic is that because examples are built from visual-numerical primitives, induction of rational errors will be based on them as well. By using this approach, VanLehn explains 85% of second through fifth graders' subtraction errors as originating from induction. VanLehn, however, admits that such an analysis is a "liberal" one, because it does not show a direct link between the examples students receive in the instructional prefixes and the rational errors they produce. Thus, it does not test the induction hypothesis directly and "conservatively" enough.

The present study argues that in order to perform a conservative evaluation of the induction hypothesis better, a new methodology is in order. Such a methodology would have more empirical control over students' acquisition of core procedures and encounters of impasses. As it stands, VanLehn tests students who are at different stages in the subtraction curriculum. He then aggregates the subtraction test results across all students, and only then analyzes the students' rational errors. There are two problems with this methodology: (a) It does not consistently test students after they have learned a certain set of examples and before they learn the following set. Therefore, it has difficulties isolating the role of a particular set of examples in the induction process. (b) This methodology does not enable one to have control over examples in "sub-lessons" that are not explicitly mentioned as independent instructional units in textbooks, but that are nevertheless emphasized in class, and can thus serve as possible sources of induction as well.
For instance, examples illustrating the rule $N - 0 = N$ may be emphasized but not taught as a separate lesson.

A NEW METHODOLOGY FOR STUDYING
THE ORIGIN OF RATIONAL ERRORS:
A COMBINED EXPERIMENTAL AND COMPUTATIONAL APPROACH

This study offers a new methodology for examining the hypothesis that students’ rational errors are created by induction. The new methodology’s strength lies in combining experimental and computational approaches. The first step is a controlled empirical investigation of induction. Specifically, participants were taught a new number system called NewAbacus, which is a written modification of the abacus. After the initial instruction of the NewAbacus number representation, participants were divided into different groups. Each group received an example of a certain part of the NewAbacus addition algorithm. Next, participants were given a range of addition problems in NewAbacus, where some were of a familiar type and some were new.

The experimental investigation of the data is primarily concerned with the question of whether participants who are exposed to the same examples will produce a set of similar errors. Given a participant’s knowledge of the NewAbacus addition algorithm (as illustrated by the examples he or she received) it was possible to control the exact points where the participant would reach impasses on new problems. If a participant uses induction from examples to overcome the impasse and get “unstuck,” then one should see a pattern where participants in a particular example condition are more likely to generate similar rational errors in response to the same impasse. Such a pattern would empirically show that the different example conditions, which result in different impasses, generate a specific family of errors, and would thus empirically support the induction-from-examples hypothesis.

The computational investigation provides a more formal and fine-grained analysis of the processes underlying the production of rational errors. By using computational modeling in LISP, it will be shown that a family of repairs (termed a “repair group”) is best modeled by modifying the procedure for the example condition it is primarily found in. The idea is to show computationally, and not only empirically, that a specific group of rational errors can be reconstructed by systematically modifying a particular example procedure. Clearly, the computational modeling is a formal abstraction (versus a literal representation) of the actual mental processes underlying the production of rational errors.

This combined empirical and computational approach takes a closer look at induction and its role in students’ production of rational errors than has previously been done. This methodology also allows one to pair the relative roles of current examples with other possible sources of induction,
the primary ones being examples of prior knowledge algorithms (i.e., base-10 multi-column addition algorithm). So far I have discussed the general framework for testing the induction hypothesis. Next, I turn to discuss the present study's hypotheses in more detail.

The first hypothesis predicts that individuals will perform more accurately on problems for which they receive worked-out examples. In essence, this is a "control" procedure for assessing whether or not students can master the specific part, or prefix, of the NewAbacus addition algorithm, as intended. It is important that they master their particular prefix, because it provides the necessary (but not sufficient) condition for induction.

The second hypothesis predicts that participants who receive the same type of worked-out examples will produce a set of similar rational errors. Similar errors are defined to be algorithmic variations of one another. For instance, one set of worked-out examples participants receive illustrates how to correctly carry the digit 6. It is predicted that when participants in this condition reach impasses on new problems they will produce a variety of illicit carries of 6 in response (a detailed discussion of performing addition in NewAbacus is presented below).

The third hypothesis is computational in nature, and provides a more formal explanation for the induction hypothesis. It predicts that participants' rational errors can be modeled best by modifying the correct procedure for the set of worked-out examples participants received. For example, illicit carries of the digit 6 can be traced computationally to the example condition that illustrates how to correctly carry the digit 6.

The computational modeling involves three steps. The first step simulates the entire NewAbacus addition algorithm. The second step models each example-type algorithm by deleting rules from the entire NewAbacus addition algorithm. The third and final step reconstructs participants' specific sets of rational errors by introducing deletions and additions to the particular example-type algorithms.

If one were to introduce all possible additions and deletions to the example algorithms one could create an arbitrarily large set of rational errors. Only a small subset of these errors would match participants' specific rational errors. Therefore, the kind of errors rather than the exact errors that participants will produce is specified a-priori. That is, this study predicts that errors would be systematically based on examples, but refrains from predicting the exact set of errors that participants will produce. The question of why participants produce only a subset of all theoretically possible errors is a topic for a different study.

Note that the procedure that remains after the deletions corresponds to what VanLehn calls a "core procedure." I call it an "example type" or an "example-type algorithm" depending on the context. The example-type algorithm correctly solves a particular set of worked-out examples.
Finally, the fourth hypothesis suggests that prior knowledge algorithms will also serve as a basis for induction of rational errors. Thus, it makes an explicit distinction between induction from current examples and induction from prior-knowledge algorithms. An interesting question arises from the extent to which current knowledge in a domain as opposed to previous knowledge in a different domain serves as a basis for induction. This question will be investigated by looking for similarities between participants’ erroneous algorithms and the multicolumn base-10 addition algorithm. The multicolumn base-10 addition algorithm is the prior-knowledge algorithm of choice because it serves as a basis for learning New Abacus addition. It is predicted that rational errors that are induced from base-10 knowledge will pervade across participants in the different example conditions, because it is shared by all participants.

Before I proceed, I would like the reader to become familiar with the NewAbacus number representation, as well as performing addition with NewAbacus numbers. The rest of the introduction is dedicated towards achieving this aim.

THE NEWABACUS SYSTEM

I have constructed a new number system that builds on the base-10 abacus system, which is a written version of number representation on an actual abacus. The base-10 abacus system represents each digit in a base-10 number by two digits, as follows. The left digit is either 5 or 0, and the right digit ranges from 0 through 4. The base-10 digit is represented by the sum of left and right digits. For example, 7 in base-10 corresponds to 52 in base-10 abacus. For larger numbers, this simple transformation holds as well (e.g., 849 = 530454 in base-10 abacus).

The base-10 abacus introduces only a few modifications to the regular base-10 multicolumn addition algorithm. Therefore, in order to create a more fertile ground for producing rational errors, I modified the base-10 abacus system still further. This new system is referred to as the NewAbacus system (see Figure 1). Similarly to the base-10 abacus system, the NewAbacus system represents each base-10 digit as two digits. However, the left digit is now either 6 or 0, and the right digit ranges from 0 through 5. As before, the base-10 digit is represented by the sum of left and right digits. For example, 7 in base-10 corresponds to 52 in base-10 abacus. For larger numbers, this simple transformation holds as well (e.g., 849 = 530454 in base-10 abacus).

The base-10 abacus introduces only a few modifications to the regular base-10 multicolumn addition algorithm. Therefore, in order to create a more fertile ground for producing rational errors, I modified the base-10 abacus system still further. This new system is referred to as the NewAbacus system (see Figure 1). Similarly to the base-10 abacus system, the NewAbacus system represents each base-10 digit as two digits. However, the left digit is now either 6 or 0, and the right digit ranges from 0 through 5. As before, the base-10 digit is represented by the sum of left and right digits. For example, 7 in base-10 is equivalent to 61 in NewAbacus (6 + 1 = 7). Note that although 64 and 65 in NewAbacus sum up to be 10 and 11 in base-10 respectively, they are illegal in the NewAbacus system. The rule that each base-10 digit must be represented by two digits in NewAbacus is violated by 64 and 65. The correct representation for 10 and 11 in NewAbacus is 0100 and 0101, respectively. Thus, there is a one-to-one mapping between the NewAbacus and base-10 number systems.
Addition in NewAbacus

The NewAbacus addition algorithm can be divided into four main parts. They are, no carry, carry into the 6 digit, carry from the 6 digit and carry into and from the 6 digit. Each part is illustrated by a set of examples (see Table 1). Let us examine each example individually. In the no-carry example, there is no difference between the base-10 and the NewAbacus addition algorithms. Addition is simply done column by column. In the carry into the 6 digit example, adding column by column produces an intermediate solution where the right digit in a pair is equal to or greater than 6. In order to correct this violation, one should carry the 6 to the left and leave the remainder. For example, when the right column in the intermediate solution is 8, one should carry a 6 and leave a remainder of 2. Note that Carrying a 6 only occurs within a pair of NewAbacus digits.

In the carry from the 6 digit example, a carry of a 1 is required between pairs. This results from the fact that two 6s are added in one column. The sum is of course 12, and thus, one should carry a 1 to the next pair, and leave a remainder of 2. Because the 2 remains in the left-digit, it violates the left-digit rule (it can only be 6 or 0). In order to correct the violation the 2 is summed with the right digit, to form a valid NewAbacus pair. Finally, the carry into and from the 6 digit example is simply a combination of the last two cases. Thus, it is the most complete algorithm out of all the example types.
### METHOD

**Participants**
Participants were 80 Yale undergraduates. They consisted of 48 females and 32 males. Some participated for a monetary reward of $5 per session, others participated as a requirement for an introductory psychology course.

**Materials**

*General-Instruction Sheet.* The General-Instruction Sheet informs subjects that they will be introduced to the NewAbacus number system. Participants were told that the NewAbacus system was a written and modified version of the abacus system, and were asked to refrain from participating in the experiment if they were familiar with adding on the abacus.

*Number-Representation Sheet.* This sheet provided a sample list of NewAbacus numbers and their base-10 equivalents. The sample list is displayed in Figure 1. The Number Representation Sheet also provided the following 4 fundamental rules of the NewAbacus number representation: (a) Each digit in base 10 was split into two digits in NewAbacus. Together they added up to the base 10 digit; (b) In each pair of NewAbacus digits the right digit ranges between 0-5; (c) In each pair of NewAbacus digits the left digit could only be either 0 or 6; and (d) 64 and 65 were not allowed to represent 10 and 11 because they violated the first rule. Each rule was followed by an example.
**Number-Representation Test.** The Number-Representation Test is a measure of understanding NewAbacus number representation. It consists of 20 problems divided into 4 sections. The test was designed to equally sample each of the four NewAbacus representation rules. The first section asked participants to give the NewAbacus equivalents of base-10 numbers (ones that were not included in the Number-Representation Sheet). For example, participants were asked to provide the NewAbacus equivalent of "200" in base-10 (answer: 020000).

The second section required participants to provide all possible NewAbacus digits for filling in missing blanks in NewAbacus numbers. For example, given "01_0," participants were expected to respond with a "6" and a "0." Conversely, the third section required them to provide NewAbacus digits that are not allowed for filling in the missing blanks of a NewAbacus number. Given "01_0," participants were expected to respond with "1" through "5".

The final section on the Number-Representation Test consisted of presenting students with a list of numbers. They were asked to decide whether or not the numbers were valid in the NewAbacus system, and if not, why. Given "1006," subjects were expected to say something like: "No, because it violates both the left-digit and right-digit representation rules. A left digit in a pair can only be 6 or 0, and a right digit in a pair can only range between 0 and 5."

**Worked-Out Examples Sheet.** There were four versions of this sheet. Each version consists of worked-out examples that illustrated only one part of the NewAbacus multi-column addition algorithm. The four types of worked-out examples are illustrated in Table 1.

**NewAbacus-Addition Test.** This test is designed to measure participants' performance on NewAbacus multi-column addition. The test is divided into 5 groups of 4 problems each, totaling 20 problems. Four groups of problems corresponded to the four example-type algorithms, and the fifth group was new to all participants (see design section which follows). There were four versions of the NewAbacus-Addition Test, each consisting of a different random ordering of the same 20 problems.

**Information Sheets.** The Information Sheets asked participants for their gender, major, SAT math score, number of college mathematics courses taken, most advanced mathematics courses taken, and grades received for each mathematics course taken. Participants were asked to rate on a 1 to 7 Likert-type scale how comfortable they were with mathematics (where 1 indicates the highest comfort and 7 the lowest comfort), to rate their mathematical ability (where 1 indicates a very high ability and 7 a very low
ability), and to express the degree of anxiety they felt during the experiment (where 1 indicates the highest anxiety and 7 the lowest anxiety).

**Procedure**

Participants first received the General-Instruction Sheet which informed them that they would be introduced to the NewAbacus number system. A Chinese subject admitted to knowing addition on the abacus, and therefore voluntarily withdrew from the study.

Next, participants were given the Number-Representation Sheet. After participants finished examining the representation rules, they were asked to verbally explain them to the experimenter in order to: (a) give participants practice with the representation rules, and (b) examine whether or not participants had understood the NewAbacus representation. Participants were told that they could refer back to the Number Representation Sheet at any point during the experiment, to ensure that they would not commit errors on later problems as a result of either forgetting the numbers or becoming too stressed.

Participants were then presented with the Number-Representation Test. This test was not only a means for assessing participants' understanding of the NewAbacus number representation, but with an opportunity for them to practice with NewAbacus numbers and gain a better understanding of the system. Only participants who received a score of 85% (out of 100%) or above were analyzed. The reason was to ensure that participants' erroneous algorithms on the final NewAbacus-Addition Test did not originate from a basic misunderstanding of number representation. The specific cut-off point of 85% was decided on because it implied that participants could get at most three problems wrong (out of 20). Since there were five problems per representation rule, they could not have all been missed. Thus participants who misunderstood an entire rule were not included. Six participants received scores that were less than 85% on the Number-Representation Test, and were therefore dropped from the analysis. Thus, initially there were 86 subjects and the final analysis was conducted on 80 subjects.

After completing the Number-Representation Test, participants were given the Worked-Out Examples Sheet. Participants were asked to examine the worked-out examples and understand them to the best of their abilities. They were then asked to verbally simulate each step in the algorithm. The verbal simulation gave participants practice with the specific part of the NewAbacus addition algorithm to which they were introduced, as well as ensured that they could actually carry out the addition.

Next, participants were asked to complete the NewAbacus-Addition Test. Simultaneously, with working out the problems on paper, participants were required to state out loud each step in the problem-solving process.
Participants' problem-solving steps were recorded on an audio cassette. During the recording, participants were instructed to describe the actual steps they were taking (e.g., "a 1 plus a 2 makes a 3"), without explaining why they were taking these particular steps. As Simon and Ericsson (1993) note, such a protocol best reflects participants' "raw" problem-solving steps by minimizing the interference of introspection.

The verbal protocol method allows a step-by-step analysis of the action sequence in every problem. For example, it solved ambiguities in the written solution when participants forgot to write down scratch marks for carries. It also prevented participants from converting the NewAbacus numbers into their base-10 equivalents, and simply doing the entire addition in base-10. Participants were explicitly told not to perform this "trick" in order not to trivialize the addition. Participants could not "cheat," because the experimenter was in the room during the protocol session. Finally, after participants completed the problem set, they were asked to complete the Information Sheets.

**Design**

The NewAbacus study employs a $4 \times 5$ example type by problem type design. Example type is a between-subjects variable. Participants were evenly divided into the four example-type groups, so there were 20 subjects per group. Participants in each of the four example-type conditions received worked-out examples that illustrated only one part of the NewAbacus addition algorithm. Specifically, example types 1-4 corresponded to the *no carry*, *carry into the 6 digit*, *carry from the 6 digit* and *carry into and from the 6 digit* cases, respectively (see Table 1). Participants were asked to solve all levels of the problem-type variable (a within-subjects variable). Problem type 1 through 4 correspond exactly to example type 1-4 (i.e., example types 1-4 illustrated how to solve these problems in a step-by-step fashion). In addition, a fifth and new problem type was introduced as well. Problem type 5 required participants to convert the illicit 64 and 65 numbers in the intermediate solution into valid representations (i.e., 0100 and 0101, respectively). For example, if the intermediate addition results in a pair that contains a 65, then the participant had to leave a 01 and carry a 1 to the right digit of the next pair, as follows:

\[
\begin{array}{c}
1 \\
020061 \\
+010004 \\
65 \\
030101
\end{array}
\]

Thus, all participants had to solve four new problems and one familiar type of problem. Note however, that example types 1-4 were ordered in
terms of increasing completeness of the NewAbacus addition algorithm. That is, each example involved more steps than the previous one.

RESULTS AND DISCUSSION

The organization of this section is as follows: It starts by providing log-linear analyses of the accuracy data. First, the accuracy of participants' solutions to the problem set was analyzed across the different example-type groups. Specifically, a particular theoretically-driven log-linear model was shown to fit the accuracy data best. Second, an analysis was done to identify which specific rational errors were found primarily in particular example conditions. This analysis was carried out by identifying positive standardized deviates (i.e., standardized square roots of the $\chi^2$ statistic). This kind of analysis enables one to ascertain whether particular examples generate specific rational errors, and thus to provide empirical support for the induction hypothesis.

Next, the induction process was analyzed by using computational modeling. The first step provides a pseudo-code that showed how the entire NewAbacus addition algorithm was simulated. The second step provides descriptions of how each example-type algorithm was modeled. Finally, the third step showed that subjects' rational errors can be reconstructed best by modifying the example procedure in which they were primarily found. The results not only provided computational support for induction from current examples, but also revealed similarities between participants' erroneous algorithms and the prior base-10 addition algorithm.

Log-Linear Modeling

Fitting a Model for Accuracy: Participants Perform Best on Problems for Which They Receive Worked-Out Examples. A particular interaction model of participants' accuracy on the problem set was specified. It was composed of three interdependent predictions. The first prediction suggested that participants would perform best on problems for which they received worked-out examples. Such a result is important because learning one's examples of the NewAbacus addition algorithm is a necessary condition for induction to occur. That is, in order to induce from an example, a person has to master it first. The second prediction suggested that participants would perform the second best on problems that required fewer steps than their worked-out example illustrated. Remember, example-type conditions 1–4 were ordered in ascending number of steps required to complete the addition algorithm (see Table 1). Thus participants in example conditions that illustrated more steps were predicted to do well on problems that
required fewer steps. Finally, the third prediction proposed that participants would do worse on problems that required more steps than their worked-out example illustrated.

This set of predictions can be quantitatively tested by fitting a log-linear model to the example type by problem type matrix. The main reason for using a log-linear analysis is that the data are categorical, that is, the cells of the matrix contain frequencies of the correct algorithms (i.e., algorithms illustrated by the worked-out examples). Performing a log-linear analysis allows one to test a-priori a specific theoretically-driven model. The predicted model is $e, p, m$ (where $e = $ example type, $p = $ problem type, $m = $ model weights). Fitting a log-linear model is analogous to performing an analysis of variance where $e$ is the main effect for the example type, $p$ is the main effect for the problem type, and $m$ is the interaction term. The advantage of performing a log-linear analysis on the accuracy data is that it not only tests the predicted model's fit with the data but pairs the likelihood of the predicted model with simpler alternative models. Thus, if two models fit the data, and one wants to choose the better fitting model, one can compute the difference between the model's fit (i.e., $\Delta G^2$), and test to see whether the difference is significant. In sum, performing a log-linear analysis allows one to test differences between competing models in a refined way. For a fuller discussion of log-linear analysis, the reader is referred to Wickens (1989).

The observed frequencies and percentages of the correct algorithm can be seen in Table 2. Please note that in each cell of the example type by problem type matrix the top number reflects the frequency of correct algorithms. The maximum frequency in each cell is 80 because there are 20 participants in each condition and 4 problems per category in the NewAbacus Addition Test.

The interaction weights are specified as follows: The diagonal cells (i.e., cells $x_{ij}$ where $i = j$) contain the highest frequencies (first prediction), cells in the upper triangle, (i.e., cells $x_{ij}$ where $i < j$) contain the next greatest frequencies (second prediction), and finally, cells in the lower triangle (i.e., cells $x_{ij}$ where $i > j$) are the least populated (third prediction). Note that problem type 5 is part of the lower triangle because it is new to all participants and is expected to generally produce incorrect responses. The $e, p, m$ model fit the data as predicted, $G^2 (10, N = 659) = 10.91$, $p > .36$. Note that in log-linear modeling nonsignificance is desired because it means that the specified model explains the significant portion of the variance. In essence, the hypothesized model corresponds to the null hypothesis, and thus the goal is not to reject the null hypothesis.

---

2 Note that the correct algorithms are different from valid alternatives. The former are the algorithms illustrated by the worked-out examples. The latter, on the other hand, are participant-created algorithms that provide valid alternatives for solving the addition problems. The valid alternatives are discussed in a special section which follows.
TABLE 2
Observed Frequencies and Percentages of the Correct Algorithm

<table>
<thead>
<tr>
<th>Example Type</th>
<th>Problem Type</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>Totals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>69</td>
<td>53</td>
<td>67</td>
<td>70</td>
<td>259</td>
</tr>
<tr>
<td></td>
<td></td>
<td>86%</td>
<td>66%</td>
<td>84%</td>
<td>88%</td>
<td>81%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td>60</td>
<td>47</td>
<td>54</td>
<td>166</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6%</td>
<td>75%</td>
<td>59%</td>
<td>68%</td>
<td>52%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>66</td>
<td>42</td>
<td>113</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1%</td>
<td>5%</td>
<td>83%</td>
<td>53%</td>
<td>35%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>45</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1%</td>
<td>3%</td>
<td>9%</td>
<td>56%</td>
<td>18%</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>17</td>
<td>12</td>
<td>23</td>
<td>14</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21%</td>
<td>15%</td>
<td>29%</td>
<td>18%</td>
<td>21%</td>
</tr>
<tr>
<td></td>
<td>Totals</td>
<td>93</td>
<td>131</td>
<td>210</td>
<td>225</td>
<td>659</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23%</td>
<td>33%</td>
<td>53%</td>
<td>56%</td>
<td>41%</td>
</tr>
</tbody>
</table>

Importantly, simpler theoretically driven models were rejected. The basic independence model \((e, p)\) predicts that there is no relationship between being exposed to a certain example type and the degree of accuracy on the problem set. It was found to be highly significant, and was therefore rejected, \(G^2 (12, N=659)=201.84, p<.001\). The model where the diagonal is more populated than the rest of the matrix \((e, p, d)\), predicts that people perform better on problems for which they receive worked-out examples, and that there is no difference between the performance of individuals who receive examples with varying degrees of completeness. It was also found to be highly significant, \(G^2 (11, N=659)=64.66, p<.001\). Finally, the model that paired the upper triangle and diagonal versus the lower triangle \((e, p, t)\) almost reached nonsignificance, \(G^2 (11, N=659)=20.06, p<.05\). This model predicts that individuals will perform equally well on problems for which they receive worked-out examples and on problems that require fewer
Specific Repair Groups are Found in Particular Example Conditions: The Induction Hypothesis (Part I). The aim of this section is to show that each repair group is found primarily in a particular example-type condition. This pattern provides empirical support for the induction hypothesis. That is, it shows that the different example conditions, which result in different impasses, generate specific groups of repairs.

Individual repairs were grouped into 11 "repair groups." Membership of a single repair in a repair group was determined by the criterion of whether it was a slight algorithmic variation on other members in the group (see Appendix A). The following are the repair groups: Base-10, Carry-6, Insert-sum, Insert-num, Comb-6, Convert-6, Carry-remainder, Elimination, 2-in-left-digit, Sum-invalid-digits, and Carry-5 (see Appendix B for details).

Specific repair groups were found primarily in particular example conditions, as predicted by the induction hypothesis. Table 3 presents the observed frequencies of repair groups by example type.

<table>
<thead>
<tr>
<th>Example Type</th>
<th>Repair Group 1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elimination</td>
<td>30</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Carry-remainder</td>
<td>20</td>
<td>9</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>Carry-6</td>
<td>8</td>
<td>94</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>Comb-6</td>
<td>17</td>
<td>54</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Insert-sum</td>
<td>17</td>
<td>16</td>
<td>57</td>
<td>7</td>
</tr>
<tr>
<td>2-in-left-digit</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Base-10</td>
<td>47</td>
<td>32</td>
<td>26</td>
<td>86</td>
</tr>
<tr>
<td>Convert-6</td>
<td>19</td>
<td>9</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>Insert-num</td>
<td>30</td>
<td>29</td>
<td>11</td>
<td>15</td>
</tr>
</tbody>
</table>

steps than their examples illustrate. However, the predicted model (e, p, m) still has a better fit than (e, p, t) since it is nonsignificant, $G^2 (10, N=659) = 10.91, p > .36$, and it reduces $G^2$ significantly, $G^2 (1, N=659) = 9.15, p < .01$. 
The induction hypothesis was statistically supported by identifying positive standardized deviates (e.g., standardized square roots of the $\chi^2$ statistic) in the repair group by example type matrix (see Table 4). Standardized deviates show the discrepancy between the expected and observed frequencies for a particular cell. The present analysis is interested in finding positive standardized deviates in each row of the repair-group by example type matrix because they indicate that a cell is more populated (i.e., has an excess frequency) than would be expected by chance. For a fuller discussion of standardized deviates see Wickens (1989).

As can be seen in Table 4, Elimination and Carry-remainder loaded positively on example type 1. Carry-6 and Combination-6 loaded positively on example 2. Insert-sum and 2-in-left-digit loaded positively on example 3. Finally, the Base-10-representation group loaded positively on example 4, all $p$'s < .01.

It should be noted that Insert-num and Convert-6 were distributed equally across all the example conditions. Insert-num is an interesting case of induction from shared knowledge of the NewAbacus number representation. Repairs in this group are characterized by the fact that participants simply

<table>
<thead>
<tr>
<th>Example Type</th>
<th>Repair Group</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elimination</td>
<td>7.0 **</td>
<td>-3.0</td>
<td>-2.3</td>
<td>-1.7</td>
<td></td>
</tr>
<tr>
<td>Carry-remainder</td>
<td>2.8 **</td>
<td>-0.5</td>
<td>1.0</td>
<td>-3.3</td>
<td></td>
</tr>
<tr>
<td>Carry-6</td>
<td>-4.5</td>
<td>10.1**</td>
<td>-3.1</td>
<td>-2.5</td>
<td></td>
</tr>
<tr>
<td>Comb 6</td>
<td>-0.3</td>
<td>8.3 **</td>
<td>-3.6</td>
<td>-4.3</td>
<td></td>
</tr>
<tr>
<td>Insert-sum</td>
<td>-1.5</td>
<td>-1.7</td>
<td>6.7 **</td>
<td>-3.5</td>
<td></td>
</tr>
<tr>
<td>2-in-left-digit</td>
<td>-2.0</td>
<td>-2.0</td>
<td>6.0 **</td>
<td>-2.0</td>
<td></td>
</tr>
<tr>
<td>Base-10</td>
<td>-0.1</td>
<td>-2.3</td>
<td>-3.1</td>
<td>5.5 **</td>
<td></td>
</tr>
<tr>
<td>Convert-6</td>
<td>1.8</td>
<td>-1.0</td>
<td>-3.3</td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td>Insert-num</td>
<td>1.9</td>
<td>1.7</td>
<td>-2.2</td>
<td>-1.4</td>
<td></td>
</tr>
</tbody>
</table>

** $p<.01$.
insert a NewAbacus number in the answer while disregarding any magnitude violation that results from this action. Convert-6, is an example of prior-knowledge interference, and will be discussed in the section on induction from prior knowledge which follows. Also, note that Carry-5 and Sum-invalid-digits are not included in Table 4 because they are too infrequent to produce a sound analysis. Details about these repairs are available from the author upon request.

The induction from examples hypothesis was thus supported empirically. In the next section, the induction process is analyzed more formally by using computational modeling in LISP. Specifically, it will be shown that a family of repairs is modeled best by modifying the procedure for the example condition in which it is primarily found.

Computational Modeling: The Induction Hypothesis (Part 2)
The hypothesis that rational errors are generated by induction is supported by using computational modeling. First, pseudo-code is provided that shows how the entire NewAbacus addition algorithm was simulated. Second, the modeling of each example-type algorithm is described. The modeling is done by deleting rules (or parts thereof) from the procedure for the entire addition algorithm. Finally, and most importantly, I show that participants' rational errors can be reconstructed by modifying the example procedure in which they are primarily found. Please note that the fact that individual repairs in a repair group can be modeled from the same example algorithm is not surprising (i.e., because they were grouped by similarity to begin with). Thus the analysis is performed on a group rather than on an individual repair level. The crucial claim is that particular repair groups are induced from specific example-type algorithms.

The results of the computational modeling not only provide computational support for the induction from current examples hypothesis, but also show the effects of the prior base-10 addition algorithm on induction of errors. The latter analysis builds on uncovering similarities between subjects' erroneous algorithms and the base-10 addition algorithm.

The algorithm for NewAbacus addition is composed of two primary phases. The first phase is responsible for adding column by column and producing an intermediate solution. (Note that carries are performed in this phase). The second phase performs tests on each pair of numbers in the intermediate solution, checking whether it violates the NewAbacus number representation. Detection of violations lead to further corrective actions until a valid solution is achieved.

When participants do not know how to proceed in either phase of the algorithm they reach an impasse. An impasse can occur either because the participant does not know what to do next or has detected a violation that resulted from the participant's previous actions (e.g., an invalid number
representation such as 08). Each participant will reach a specific impasse depending on the example condition to which he or she is assigned, and the problem type on which the participant is working. The induction hypothesis suggests that the participant chooses to apply a particular repair based on the example he or she receives. These repairs are modeled by using procedures in LISP.

As a first step for simulating participants' production of rational errors, the correct algorithm for NewAbacus addition was modeled. The pseudocode for this algorithm is presented in Figure 2. This representation is a hybrid of a procedural and a production-system architecture. The procedural aspects highlight the overall structure of the algorithm. The production-system components specify portions of the algorithm in terms of conditions (C) and actions (A). The structure of the condition-action rules can be illustrated by the Right-carry-6 rule: If the right digit is equal to or greater than 6 (C) then carry a 6 into the left column (A).

Next each example-type condition was modeled by deleting rules (or parts thereof) from the correct NewAbacus addition algorithm. The assumption was that participants were only able to solve problems that were illustrated by their example type. Thus, the subset of rules from the general NewAbacus addition algorithm that corresponded to the worked-out examples was retained. Rules that were not demonstrated by the worked-out examples were deleted. For instance, the example 2 algorithm was produced by only retaining the Right-no-carry, Left-no-carry, Left-carry-1, and Correct-carry-6 rules, as well as the condition rules for Correct-carry-2, Correct-carry-64, and Correct-carry-65. The action rules for the Correct-carry-2, Correct-carry-64, and Correct-carry-65 were deleted. The other example algorithms were modeled in a similar fashion.

Note that all participants had the condition rules (e.g., the right digit can only range between 0 and 5) for the number representation, because they came from the initial instruction phase. Participants, however, may not necessarily have had all the action rules for correctly rectifying violations of number representations (e.g., if the right digit is greater than 5, carry a 6 to the left digit and leave the remainder).

It is also important to note that the condition-action rules comprise the part of the algorithm that varies across subjects in the different conditions. The overall procedural structure was assumed to be shared by all participants. The general structure refers to adding all columns from right to left, and distinguishing between right and left digits in a pair.

**Induction of Rational Errors.** In order to support the induction from examples hypothesis computationally, it will be shown that algorithms that simulate participants' production of rational errors can be formally reconstructed from the example type participants received. Specifically, it can be
NewAbacusAddition (top bottom)
  AddNumbers (top bottom)  \textit{creates an intermediate result}
  SatisfyConstraints (intermediate_result)  \textit{produces the final result}

AddNumbers (top bottom)
  \textbf{while} (column is not empty)
    AddColumn (top_digit bottom_digit carry_digit)
    get next column

AddColumn (top_digit bottom_digit carry_digit)
  \texttt{digit\_sum} = top_digit + bottom_digit + carry_digit

\textbf{Right-digit-rules}
  \textbf{if} working on a right digit \textbf{then}
  \begin{align*}
  \text{if } (\text{digit\_sum} < 6) \text{ then} & \quad \text{AddToIntermediateResult (digit\_sum)} \\
  \text{else if } \text{digit\_sum}  < 10 \text{ then} & \quad \text{Carry-6 to left column of current pair} \\
  & \quad \text{AddToIntermediateResult (digit\_sum - 6)} \\
  \text{else} & \quad \text{Carry-1 to right column of next pair} \\
  & \quad \text{AddToIntermediateResult (digit\_sum - 10)}
  \end{align*}

\textbf{Left-digit-rules}
  \textbf{else}
  \begin{align*}
  \text{if } ((\text{digit\_sum} = 0) \text{ or } (\text{digit\_sum} = 6)) \text{ then} & \quad \text{AddToIntermediateResult (digit\_sum)} \\
  \text{if } (\text{digit\_sum} = 12) \text{ then} & \quad \text{Carry-1 to right column of next pair} \\
  & \quad \text{AddToIntermediateResult (digit\_sum - 10)}
  \end{align*}
SatisfyConstraints (intermediate_result)
   while (pair is not empty)
      FixPair (left_digit right_digit)
      go to next pair

Correcting violations

FixPair (left_digit right_digit)
   repeat
   Correct-carry-6
       if (right_digit > 6) then
           right_digit = right_digit - 6
           Carry-6 to left_digit
   Correct-carry-2
       if (left_digit = 2) then
           left_digit = 0
           Carry-2 to right_digit
   Correct-carry-1
       if (left_digit ≥ 10) then
           left_digit = left_digit - 10
           Carry-1 to right column of next pair
   Correct-64-rule
       if ((left_digit = 6) and (right_digit = 4)) then
           left_digit = 0
           right_digit = 0
           Carry-1 to right column of next pair
   Correct-65-rule
       if ((left_digit = 6) and (right_digit = 5)) then
           left_digit = 0
           right_digit = 1
           Carry-1 to right column of next pair
   until pair is valid i.e., until no rules have applied
AddPairToFinal Result (pair)

Figure 2. Pseudo-code for the NewAbacus addition algorithm.
demonstrated that a group of repairs is modeled best by modifying the example algorithm in which it is primarily found. Notice that the term best but not the term only is used for this kind of modeling. That is, it is possible to modify many example algorithms in order to recreate a family of repairs. The best-fitting procedure, however, is the one that requires the least number of modifications.

For example, the Carry-6 repair group was primarily found in example 2. It can be computationally shown that Carry-6 repairs are produced by deleting and adding rules to the example 2 algorithm. For example, the repair leave-a-6-carry-a-6-as-a-ten can be simulated by deleting the Left-Carry-1 rule (a prior-knowledge rule), and inserting a Carry-6 rule instead. Remember, this repair occurs on problems that require adding two 6s in a column. Participants who use it leave a 6 in the left column and carry a 6 into the right column.

The Leave-a-6-carry-a-6-as-a-ten repair is a clear example of overgeneralization from the condition 2 example-type algorithm. Condition 2 illustrates how to carry a 6 within pairs. Participants using the Leave-a-6-carry-a-6-as-a-ten repair generalized the example to mean that 6s should also be carried between pairs. Other Carry-6 repairs demonstrate clear overgeneralization as well. A particularly extreme case was illustrated by a participant who decided that one should forever carry a 6, and therefore concluded that the answers to the problems in the problem set were infinite! Similar simulations of how other repairs are modeled by modifying the example algorithm they are primarily found in can be obtained from the author by request.

Please note that about 98% of participants' errors were found to be rational\(^3\) (754 out of 765 errors). Only a few errors were undiagnosed due primarily to the fact that a small number of students did not verbalize well during the protocol recordings and produced errors that could not be easily categorized. This figure is much higher than VanLehn's (1990), who found that only 55% of errors could be explained as being generated by buggy procedures (i.e., rationally). The reason for the difference between the figures is methodological, Because the current study used protocol analysis it was possible, for the most part, to identify the exact bug that produced a rational error. VanLehn, on the other hand, only had answer data and could not always ascertain which bug, out of a possible set of bugs, produced a particular rational error. The higher percentage of rational errors accounted for by the current study is largely due to the fact that the current analysis is item-based (i.e., the unit of analysis is a problem) whereas VanLehn's analysis was student-based (i.e., the unit of analysis was the student's entire test).

\(^3\) Some of the rational errors include "slips" (i.e., careless mistakes such as saying "3" out loud but writing down a "2"). Because the analysis was based on verbal protocols, it was possible to separate a slip from the underlying buggy procedure that produced the rational error.
The next subsection shows that participants' rational errors are not only induced from current examples, but build on prior knowledge as well.

*Repairs that Bear Superficial Similarity to the Base-10 Addition Algorithm: Manifestations of Prior Knowledge.* Participants' rational errors were also traced back to their multicolumn base-10 addition algorithm. The interference of prior base-10 knowledge is expressed in two types of repair groups. The first is a "passive" family of repairs. Their most obvious characteristic is that they violate a NewAbacus number representation, but are valid in base-10 (e.g., *Base-10-representation* repairs such as leaving 64, 65, 23, and 66, unchanged). These violations are passive, because they suggest that participants neglect to detect a number-representation violation rather than actively produce a rational error.

The second group of repairs shows more "active," or algorithmic base-10 interference. The salient aspect of repairs in this group is that they show superficial perceptual similarities to the base-10 multicolumn addition algorithm. Specifically, participants who create these repairs persist in applying the familiar carry of a 1, even when it is inappropriate to do so. For example, the repairs belonging to *Convert-6-into-ten* falsely convert a carry of a 6 into a carry of 1. Overall, induction from prior knowledge accounted for 24% of participants' repairs (i.e., there are 8 repairs in the *Base-10-representation* group out of a total number of 33 repairs).

Participants did not only produce rational errors, but also invented valid alternative algorithms as well. I turn to discussing these algorithms next.

*Induction of Alternatives to the Correct Algorithms: The Corpair and Pairwise Conversion Algorithms.* Interestingly enough, participants, did not confine themselves to using only the given correct algorithms (i.e., those that were illustrated by the worked-out examples) but invented two correct algorithms of their own. They are *Corpair* and *Pairwise Conversion*. The observed frequencies of the valid alternatives can be seen in the top entries of Table 5. *Corpair* was used on problems that required carries of 6. Participants who invented this algorithm first added column by column, reaching an intermediate solution. They then checked the validity of pairs in the solution. An illegitimate pair was converted into its appropriate NewAbacus representation as follows:

```
 10105
+60103
 61028
 61026
```

In the above example, the participant first added column by column leaving a "08" in the rightmost pair. She then realized that "08" was not valid,
so she replaced it with its equivalent in NewAbacus, which is a "62". This procedure is different from the correct algorithm, because the latter requires the participant to immediately carry a 6 into the left digit. Only then does the participant check the validity of each pair in the solution.

*Pairwise Conversion* is a clever "shortcut," that trivializes performing addition with NewAbacus numbers. The participants who use this algorithm do not add column by column. Instead, they add pair by pair. First, they convert the top and bottom pairs in the problem into their base-10 equivalents (e.g., 03 = 3 and 04 = 4). Then, they sum up the numbers in base-10 (e.g., 3 + 4 = 7). Finally, they convert the sum back into the NewAbacus number representation (e.g., 7 = 61), and proceed to write it in the answer. This procedure is repeated on a pair by pair basis. The following is an example of *Pairwise Conversion*:

```
  01  02  03  04
+ 03  01  04  01
  04  03  61  7
```

An analysis of standardized deviates revealed that *Corpair* was primarily generated by participants in conditions 1 and 3, who were not shown how to carry a 6 (see bottom entries in Table 5). By using *Corpair* subjects bypassed the need to carry a 6, by first adding column by column and only then attempting to satisfy the constraints of the new system. *Pairwise Conversion* was primarily generated by participants in condition 1 who only received the no-carry examples (see bottom entries in Table 5). A possible explanation of why *Pairwise Conversion* was generated by participants who had the least knowledge of NA addition will be elaborated on in the discussion section.
GENERAL DISCUSSION

On the one hand, students are creative and active problem solvers who not only invent rules but apply them deliberately and systematically. On the other hand, the rules students create are often flawed. This study offers a possible answer to this puzzle. It suggests that "what goes wrong" in the generation of rational errors is a result of inductive reasoning processes that operate on incomplete knowledge. That is, when students encounter unfamiliar problems they do not know how to solve, they overgeneralize or overspecialize rules from familiar examples in the domain in order to produce a solution. This idea was referred to as the "induction hypothesis" (VanLehn, 1986).

This study expanded on past formulations of the induction hypothesis by arguing that induction of rational errors is not only confined to the current examples a student is exposed to in the process of learning a particular arithmetic skill, but it also builds on examples of prior knowledge algorithms as well. Thus, a student's rational errors in a particular domain (e.g., subtraction) may be a result of overgeneralizing or overspecializing rules from prior knowledge algorithms in different domains (e.g., addition).

One can argue that the idea that students use familiar examples from current or past knowledge in order to solve new problems is obvious. The surprising aspect of this finding, however, is not the fact that students use familiar examples in order to create new algorithms, but the rule-based process by which students use them. That is, the fact that students use examples as a basis for creating well-defined but erroneous procedures is an important finding.

The induction hypothesis was confirmed both empirically and computationally. On the empirical front, it was first shown that participants did better on NewAbacus addition problems for which they received worked-out examples. This finding provides the necessary (but not sufficient) condition for induction. That is, in order for a student to induce from an example, the student first has to master it.

The more significant finding was that participants who were exposed to a particular example of the NewAbacus addition algorithm, and thus reached the same impasses on novel problems, tended to generate specific families of repairs. This finding suggests a causal relation between the particular examples one receives and the specific strategies one chooses for solving new problems. It therefore supports the role of examples as providing a basis for the systematic generation of rational errors.

The computational investigation provided more formal support for induction. It showed that each example procedure could best be modified to produce a specific group of repairs. For instance, the leave-a-6-carry-a-6-as-a-ten repair (a member of the Carry-6 repair group) was simulated by delet-
ing the Left-Carry-1 rule and inserting a Left-Carry-6 rule instead, in the example 2 algorithm. By using this process it was possible to model 67% of the incorrect repair groups that were discovered. Thus, most of the rational errors that were found can be explained by induction from current examples.

Current examples were not the only sources of induction that were found. Another important source of induction was prior knowledge, namely, the base-10 multicolumn addition algorithm. Prior knowledge created both passive and active repairs. Passive repairs are most likely a result of a failure to detect a violation, rather then an active construction of a rational error (e.g., leaving pairs in the solution that are valid in base 10, but are not valid in NewAbacus such as 68 or 23). Active prior knowledge repairs, on the other hand, showed algorithmic base 10 interference. They built on the superficial (versus conceptual) properties of the multicolumn base 10 algorithm (e.g., Convert-6-into-ten falsely converts a carry of a 6 into a carry of a 1). The finding that participants commonly misused carries of 1 suggested that they acquired a strictly syntactic rather than a semantic understanding of the carrying action in base-10.

The distinction between induction from current examples and induction from prior knowledge algorithms is an important one, not only from a theoretical point of view but from an educational one as well. If a teacher cannot trace a student's rational error to an example in a particular domain, the teacher should take a step back and examine prior knowledge algorithms in other domains as possible sources. For example, it has been observed that zero-rules from the domain of addition (i.e., \(N + 0 = N\), and \(0 + N = N\)) affect students' bugs in the domain of subtraction (VanLehn, 1986). This phenomenon can be illustrated by the bug \(0 - N = N\) where students appear to overgeneralize commutativity of zero-rules from addition to subtraction.

The performance of participants in this study makes it plausible that students use induction from current examples in a domain and from past examples in a different domain for producing rational errors in the process of learning arithmetic.

Interestingly, participants not only created erroneous algorithms but also invented valid alternatives to the worked-out examples. An especially surprising finding is that Pairwise Conversion was primarily invented by participants in condition 1 who only received the no-carry examples. It thus appears that knowing the least allowed some subjects to construct valid alternatives. The explanation may be that participants in the other example conditions became too entrenched by their example type. This is akin to the "response set" phenomenon (Luchins, 1942) which suggests that learning a complex algorithm for solving a problem hinders the ability to find simpler ways of solving the same problem. However, this phenomenon may also be due to the fact that participants were instructed not to convert the NewAbacus numbers into base-10 numbers and solve the addition in base-10. It may be
that participants in condition 1 came close to violating this constraint by cleverly converting pairs of numbers, because they had no other way to proceed.

Investigating how students induce new algorithms from examples and prior knowledge gives us insight into the nature of mathematical thinking and learning in general. First, it tells us that induction is based on syntactic rather than conceptual features of current examples and prior knowledge algorithms. That is, students' erroneous algorithms build on visual-numerical features of problems while disregarding any number violations (e.g., number magnitude) that may result. Attention to syntax rather than semantics can possibly be remedied by teaching for meaning (Greeno, 1983).

Second, there is a consensus in the problem-solving literature that learning from examples is an important mechanism in the acquisition of a mathematical skill (see for example, Simon & Zhu, 1988). It is not surprising then, that under particular conditions the same inductive processes that facilitate correct learning may underlie erroneous performance, as well. The culprit is not the inductive process that goes astray, but the fact that the knowledge it builds on is incomplete. Thus, what could be a sound generalization given adequate information turns into an overgeneralization in the presence of incomplete knowledge. The idea that the same inductive mechanisms underlie correct as well as incorrect problem-solving helps explain the rational aspect of erroneous performance.

REFERENCES


**APPENDIX A**

The Process of Combining Repairs into Repair Groups

The following is a list of the eleven repair groups and a brief description of the similarities between the repairs that make up each group.

1. **Insert-num.** Repairs in this group involve inserting a NewAbacus pair whose sum is between 6 and 9 into the result, instead of performing a carry of 6.
2. **Insert-sum.** Repairs in this group involve inserting a NewAbacus pair whose sum is greater than or equal to 10 into the result, instead of performing a carry of 1.
3. **Elimination.** Repairs in this group involve eliminating digits that violate the NewAbacus number representation. For example, participants convert an intermediate illicit digit (e.g., 8) into the highest allowable digit (e.g., 5), or simply set it to 0.
4. **6-converted-to-ten.** Repairs in this group involve converting a 6 in the intermediate result into a 1, and then, carrying the 1 to the next column.
5. **Carry-a-6-as-a-ten.** Repairs in this group involve carries of 6 between pairs. Note that correct carries of 6 are valid only within a pair.
6. **Carry-remainder.** Repairs in this group involve leaving the highest possible valid digit (i.e., 5) or pair of digits (i.e., 63), and carrying the remainder to the next column or pair, respectively.
7. **Base-10 representation.** Repairs in this group involve leaving digits in the intermediate result that are valid in base-10, but are not valid in NewAbacus. In essence, these repairs are the result of a failure to detect a NewAbacus number violation.
8. *2-in-left-digit.* There is only one repair in this group. The participant who created this repair wrote a "2" in the left digit of the first pair, and carried a 1 to the next digit. He then continued to sum up the digits of the pair.

9. *Sum-across-invalid-pairs.* There is only one repair in this group. Participants who used this repair first added column by column. When they detected an illicit pair in the intermediate result, they simply added the sum of the pair to the sum of the pair on its left and inserted the result back into the solution.

10. *Carry-a-5-as-a-ten.* There is only one repair in this group. Participants who used this repair changed a 6 in the right digit by leaving a 1 and carrying a 5.

11. *Combination-6.* Repairs in this group involve a combination of a Carry-6-as-a-ten repair and a repair that belongs to a different group.

**APPENDIX B**

**Repair Groups**

The organization of the section is as follows: Each repair group’s name appears in bold print. It is followed by its individual repair members. Each individual repair’s name is followed by parenthesis. Inside the parentheses are: number of problems on which a repair appeared alone, number of problems on which it appeared in combination with other repairs, and the total number of participants who used it.

**Insert-num**

**Insert a New Abacus number once (8, 0, 7)**

\[
\begin{align*}
010401 + 600302 &= 610703 \\
&= 6106103
\end{align*}
\]

**Insert 01 instead of carry (7, 15, 5)**

\[
\begin{align*}
606160 + 020161 &= 21 \\
&= 62620103
\end{align*}
\]
Leave a 6 insert a 60 (2, 0, 2)
030063
+ 020205
  05026062

Insert number and a 0 with every digit (6, 0, 3)
036002
+ 020405
  00056004061

Insert a New Abacus number more than once (62, 0, 10)
606202
+ 020403
  626605
  62606005

**Insert-sum**

Insert sum with 0 (4, 0, 1)
026002
+ 026161
  0412163
  0401020163

Insert pair sum (93, 0, 14)
010263
+ 040302
  050565
  05050101

**Elimination group**

Leave highest single digit number eliminate remainder (3, 4, 4)
010105
+ 600103
  610208
  610205
Leave highest two digit number eliminate remainder (3, 0, 3)

\[
\begin{align*}
016105 \\
+620160 \\
636265 \\
636263
\end{align*}
\]

Eliminate two 6s when adding two 6s (3, 1, 4)

\[
\begin{align*}
616005 \\
+026202 \\
630205
\end{align*}
\]

Eliminate a digit (16, 7, 13)

\[
\begin{align*}
010263 \\
+600104 \\
610367 \\
610360
\end{align*}
\]

Eliminate a 6 carry a 6 as a ten (2, 0, 2)

\[
\begin{align*}
1 \\
026360 \\
+606102 \\
630462
\end{align*}
\]

6-converted-to ten group

carry a 6 converted to ten (14, 12, 11)

\[
\begin{align*}
1 \\
020263 \\
+600201 \\
620504
\end{align*}
\]

6 converted to ten twice (23, 3, 11)

\[
\begin{align*}
2 \\
026360 \\
+606102 \\
622462 \\
640462
\end{align*}
\]
leave a 6 carry a 6 converted to ten (7, 0, 6)

\[
\begin{align*}
&\quad \ 1 \\
&030063 \\
+&020205 \\
&050362
\end{align*}
\]

**Carry-a-6-as-a-ten group**

carry a 6 as a ten (72, 51, 31)

\[
\begin{align*}
&\quad \ 6 \ 6 \ 6 \\
&010105 \\
+&000103 \\
&60010202
\end{align*}
\]

leave a 6 carry a 6 as a ten (21, 5, 8)

\[
\begin{align*}
&\quad \ 6 \\
&030063 \\
+&020260 \\
&056263
\end{align*}
\]

**Carry 6 as ten twice (6, 0, 2)**

\[
\begin{align*}
&\quad \ 6 \\
&016202 \\
+&010403 \\
&01040005
\end{align*}
\]

Leave highest single digit number carry 6 as a ten (1, 0, 1)

\[
\begin{align*}
&\quad \ 6 \\
&020401 \\
+&010201 \\
&630502
\end{align*}
\]

carry a 6 until infinity (7, 0, 1)

\[
\begin{align*}
\infty \leftarrow &\quad \ 6 \ 6 \ 6 \ 6 \ 6 \\
&010304 \\
+&020203 \\
&\quad \ 7
\end{align*}
\]

\[
\begin{align*}
\infty \leftarrow &\quad 00030501
\end{align*}
\]
**Carry-remainder group**

Leave highest two digit number carry remainder (38, 18, 9)

\[
\begin{align*}
&1 \\
&06100 \\
+&0\ 0304 \\
&626404 \\
&626304
\end{align*}
\]

Leave highest single digit number carry remainder (0, 3, 1)

\[
\begin{align*}
&1 \\
&040401 \\
+&000201 \\
&050502
\end{align*}
\]

**Base-10 representation repairs**

07/08/09 violation (18, 0, 11)
Leaves either an 07, 08, or 09, as is (e.g., 020407)

64 violation (40, 0, 22)
Leaves 64 as is (e.g., 626401)

65 violation (48, 0, 26)
Leaves 65 as is (e.g., 010365)

Left digit violation (44, 17, 26)
Leaves a digit other than 0 or 6 in the left digit (e.g., 042361)

6 in the right digit (27, 7, 15)
Leaves a 6 in the right digit (e.g., 620605)

Omit leftmost zero (3, 0, 5)
Omits the leftmost zero in the solution (e.g., 46261)

Convert 64 to 10 (2, 0, 1)

\[
\begin{align*}
&026001 \\
+&600400 \\
&626401 \\
&621001
\end{align*}
\]
Convert sum into base-10 then sum digits (8, 0, 2)

\[
\begin{array}{c}
606104 \\
+010501 \\
616605 \\
611205 \\
610305 \\
\end{array}
\]

2-always-in-left-digit-carry-1 (16, 0, 1)

\[
\begin{array}{c}
1 \\
020202 \\
+600103 \\
620425 \\
020461 \\
\end{array}
\]

Sum-across-invalid-pairs (8, 0, 5)

\[
\begin{array}{c}
030063 \\
+020205 \\
050268 \text{ sum pairs to 16} \\
050160 \\
\end{array}
\]

Carry a 5 as a ten (3, 0, 3)

\[
\begin{array}{c}
5 \\
030104 \\
+010202 \\
040306 \\
040351 \\
\end{array}
\]